

THE “PARADOX” OF COMPUTABILITY AND A RECURSIVE RELATIVE VERSION OF THE BUSY BEAVER FUNCTION

FELIPE S. ABRAHÃO

National Laboratory for Scientific Computing (LNCC), Brazil

ABSTRACT. In this article, we will show that uncomputability is a relative property not only of oracle Turing machines, but also of subrecursive classes. We will define the concept of a Turing submachine, and a recursive relative version for the Busy Beaver function which we will call Busy Beaver Plus function. Therefore, we will prove that the computable Busy Beaver Plus function defined on any Turing submachine is not computable by any program running on this submachine. We will thereby demonstrate the existence of a “paradox” of computability *a la* Skolem: a function is computable when “seen from the outside” the subsystem, but uncomputable when “seen from within” the same subsystem. Finally, we will raise the possibility of defining universal submachines, and a hierarchy of negative Turing degrees.

1. INTRODUCTION

In first place, we must briefly introduce the ideas behind the definitions, concepts and theorems exposed in the present article. It is true that, following an interdisciplinary course of study, this work focuses on manifold inspirations coming from several different fields of knowledge. However, for present purposes it is essential to mention the two foremost: Skolem’s “paradox” and metabiology.

Skolem’s “paradox” derives from Cantor’s famous theorem on the uncountability of infinite sets, for instance of real numbers or of the set of all subsets of natural numbers, and also from Löwenheim-Skolem’s theorem on the size of models in satisfiable theories. Briefly, the “paradox” is: there is a countably infinite model for a theory (e.g. ZFC, if we accept it as consistent) that proves there are uncountably infinite sets. Therefore, when we look at the set of elements in the theory’s model which correspond homomorphically to the elements in the set that the theory demonstrates contain an

Key words and phrases. Relative uncomputability, Subrecursive hierarchies of uncomputabilities, Busy Beaver function, Paradox of computability.

This research was supported by the National Council for Scientific and Technological Development (CNPq), Brazil, as a PhD Fellowship at the Federal University of Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil.

This is a preprint of a book chapter published in *Information and Complexity*, Mark Burgin and Cristian S. Calude (Editors), World Scientific Publishing, 2016, ISBN 978-981-3109-02-5, available at <http://www.worldscientific.com/worldscibooks/10.1142/10017>.

uncountable amount of elements, may however contain a countable amount of elements. Does this contradiction represent, in fact, a paradox?

That question was answered by Skolem in 1922, being that the property of countability of real numbers depends on the existence of a function within the model that makes this bijective enumeration. This function cannot exist within any ZFC model - if that is the chosen axiomatic system - because, if it existed, it would render countable the set of all real numbers. However, it may exist when seen “from the outside”, when this function never belongs to the model. In other words, the function that “bijects” the natural onto the real numbers never belongs to any model of any Set Theory¹ - but exists nevertheless. Thus, it is understood that this does not constitute a true paradox. It forms what one may call a pseudoparadox: when seen “from the outside”, an object has a certain property, but when seen seem “from within” it has the opposite property. For this reason, it makes sense to call the Skolem’s “paradox” a pseudoparadox of countability. Therefore, one may ask the question: just as there is a pseudoparadox of countability, could there be a pseudoparadox of computability? We will address this subject in the present paper.

Metabiology is a field of theoretical computer science with a transdisciplinary “heart” that studies general principles of biological relations at a meta-level focusing on the open-ended evolution of systems and is inspired by both the theories of biological evolution and by algorithmic information theory (AIT). While the already developed and well-established fields of population genetics and evolutionary computations are driven towards simulations of evolutionary populations and statistical properties of those populations, metabiology is driven towards achieving theorems. It uses all available tools from theory of computation, algorithmic information and metamathematics to build and study abstract models – applicable or not.

Unlike the first models made by Chaitin, if we want a “nature” without access to oracles - at least, without access to real oracles - it needs to be a system that can be completely simulated on a universal Turing machine, or on a sufficiently powerful computer. This would also allow us to do experimental computer simulations on the evolution of digital organisms in the future.

In attempting to create a metabiological nature which is computable, but that “behaves in the same way” as an oracular one, i.e., in the same way as an uncomputable nature - in other words, as a hypercomputer - we face a series of difficulties. The first, highlighted in the literature, is the absence of a program that will solve the *halting problem*. This is required to determine whether or not an algorithmic mutation will result in a new organism/program (a mutation’s output given by a prior organism as input) and whether or not the organism/program has a higher fitness (its output) than the previous one. A computable nature would necessarily need to be capable of running a function capable of accomplishing this task, which we know is impossible for any arbitrary program. Note also that solving the halting problem, computing the bits of a Chaitin number Ω , and computing the Busy Beaver function, are equivalently uncomputable problems of Turing

¹ Of course, provided it is strong enough and satisfiable.

degree $\mathbf{0}'$. How would evolution, then, occur within a computable simulation of “Nature”?

However, this paper does not intend to present mathematical exercises in metabiology, but to focus on presenting the elements which allow proving a computable metabiological model: “**sub-uncomputability**”. The present paper comes from a solution we gave to the question to the above, as shown in [4]. We intend to demonstrate several fruitful properties and raise issues for future study within the mathematics of theoretical computer science.

Basically, we will call a **Turing submachine** any Turing machine that always gives an output for any input, i.e. always halts. The prefix “sub” parallels the term subrecursion. A subrecursive class is one defined by a proper subset of the set of all problems with Turing degree $\mathbf{0}$. Therefore, a **subcomputable** class of problems will be subrecursive, because it will never contain all recursive/computable problems. The term subrecursion is also used to characterize subrecursive hierarchies, as in Kleene and Grzegorzcyk, covering all primitive recursive functions. But for us, the prefix refers more specifically to the concept of subrecursive class.

Note that Turing submachine is just another terminology for **total Turing machines**. However, despite the fact that they are just different names for the same object and can be used interchangeably, the expression “total Turing machine” might not immediately capture its relevant properties related to the present paper.

Every total computable function (or total Turing machine) defines a subrecursive class which is a proper subclass of other subrecursive classes (and of the class of all recursive functions). For example, the total Turing machine will be a subsystem of another total machine which is capable of computing functions that are relatively uncomputable by the former. This very idea of being part of another non-reducibly more powerful machine – that comes from sub-uncomputability, as we will show – is the core notion of the expression “submachine”, conveying and bearing the ideas of hierarchies of subrecursive classes together with the powerful concept of Turing machines. Thus, the terminology Turing submachine emphasizes this property of total Turing machines being always able to be part of another proper and bigger machine. For more of this discussion, see item 4.

The central theme is building, or rather proving, a system (a Turing machine) that can “behave” in relation to a subsystem (its Turing submachine) in the same way as a hypercomputer (an oracle Turing machine) would behave in relation to a subsystem (in particular, a universal Turing machine). In fact, we will not emulate all - which might be impossible - the properties of a hypercomputer in relation to a computer, but focus on defining a function $BB^{+}_{P'_T}(N)$ analogous to a Busy Beaver function (in Chaitin’s work, function $BB'(N)$), so that this function will behave in relation to the Turing submachine in the “same” way as the original Busy Beaver behaves in relation to a universal Turing machine. In other words, $BB^{+}_{P'_T}(N)$ must be relatively uncomputable by any **subprogram** (a program running on a Turing submachine), the same way the original Busy Beaver is uncomputable by any program running on a universal Turing machine. This

phenomenon will be called **recursive relative uncomputability**, or **sub-uncomputability**.

2. LANGUAGE L

The first important definition that needs to be established is the very programming, or universal machine, language with which we will work. It is important to us that the submachine $U_{P',T}$ must be programmable. This language can be used on any usual computer, that is, its properties and rules of well-formation are programmable. Ultimately, this will lead us to the conclusion that the phenomenon of subcomputation may occur “within” computers as we already know them, which are universal Turing machines with limited resources.

Why concatenations? They provide a direct way of symbolizing a program taking any given bit string as input - for example, a program p that is, actually, program p' taking program p'' as input - which makes this program act as a function. Note that this type of program is already used to demonstrate the *halting problem*, or demonstrate that the Busy Beaver function is uncomputable. But the form it may assume is completely arbitrary, as a universal Turing machine, in any case, will run it. Therefore, it is no wonder we need this condition - this functionalizing special concatenation - in our language. As we are trying to build a computer that can emulate uncomputability, it is necessary that we can “teach” a machine to perform and recognize these “concatenations” within the language it is working in.

Many of the properties of the language L , below, are not required for this study; however, they were required to demonstrate the evolution of metabiological subprograms *a la* Chaitin.

To differentiate from the optimal functionalizing concatenation, which is joining strings in the most compressed way possible, provided it remains well-formulated, this special functionalizing concatenation will be denoted as “ \circ ”, while the optimal functionalizing concatenation will be symbolized as “ $*$ ”.

2.1. Definition. We say a universal programming language l , defined on a universal Turing machine U , is **recursively functionalizable** if there is a program that, given any bit strings P and w as inputs, will return a bit string belonging to l which will be denoted as $P \circ w$, whereby $U(P \circ w)$ equals “the result of the computation (on U) of program P when w is given as input”. In addition, there must be a program that will determine whether or not a bit string is in form $P \circ w$ for any P and w , and is capable of returning P and w separately. Analogously, the latter must be true for the successive concatenation $P \circ w_1 \circ \dots \circ w_k$, with program P receiving w_1, \dots, w_k as inputs.

Now, the general definition of language L can be defined as: let U be a universal Turing machine running language L , a universal language that is **binary, self-delimiting, recursive**, and **recursively functionalizable** and that there are constants $\epsilon, C \in C'$, for every P, w_1, \dots, w_k , where:

$$|w_i| < |P \circ w_1 \circ \dots \circ w_k|, \text{ for } i = 1, 2, \dots \text{ or } k$$

and

$$|P \circ w_1 \circ \dots \circ w_k| \leq C \times k + |P| + |w_1| + |w_2| + \dots + |w_k|$$

and

$$H(N) \leq C' + \log_2 N + (1 + \epsilon) \log_2 (\log_2 N)$$

3. DEFINITIONS

- (a) W is the set of all finite bit strings, where the computable enumeration of these bit strings has the form $l_1, l_2, l_3, \dots, l_k, \dots$.
For practical purposes, a language may be adopted where $l_1 = 0$.
- (b) Let $w \in W$.
 $|w|$ denotes the size or number of bits contained in w .
- (c) Let N simply symbolize the corresponding program in language L for the natural number N . For example, $P \circ N$ denotes program $P \circ w$ where w is the natural number N in the language L .
- (d) If function f is computable by program P , then f may also be called function P .

4. TURING SUBMACHINES

A key concept in the present article is the idea that a subsystem can do almost anything its system can, however, with resources limited by the very system. We follow the conventional understanding in which a computation that is a part of another computation may be called a subcomputation, and a machine that is a part of another machine may be called a submachine. In our case, a system can be taken as a Turing machine, and a subsystem can be taken as a Turing submachine. For example, a Turing submachine can be a program or subroutine that the “bigger” Turing machine runs, always generating an output, while performing various other tasks. Note that it is true (a theorem) that for every total Turing machine there is another Turing machine that completely emulates and contains the former total Turing machine, in a manner that the computations of the latter contains the computations of the former.

In fact, we are using a stronger notion of subsystem based upon this conventional notion: a subsystem must be only able to do what the system knows, determines and delimits. This way, submachines will only be those machines for which there is another “bigger” machine that can decide what is the output of the former and whether there is an output at all. Note that every machine that falls under this definition always defines an equivalent **total Turing machine** (with a signed output corresponding to the case where the former does not halt); and every total Turing machine falls under this definition.

We will use another concept of vital importance: **computation time**. Similarly to time complexity, we will call T a program that calculates how many steps or basic operations U performs when running program p . Thus, if $U(p)$ does not halt, then $U(T * p)$ will not halt either, and vice versa.

Let P_f be a program running on U defined in language L , computing a total function (a function defined for all possible input values) f such that

$f : L \longrightarrow X \subseteq W$. The language W does not need necessarily to be self-delimiting, and may be comprised of all bit strings of finite size, as long as they may be recursively enumerated in order, as l_1, l_2, l_3, \dots . For practical reasons, we will choose an enumeration where $l_1 = 0$.

A “**Turing submachine**” or **total Turing machine** U/f is defined as a Turing machine in which, for every bit string w in the language of U , $U/f(w) = U(P_f \circ w)$.

This definition is quite general and transforms any total computable function into a Turing submachine. In fact, as said in the introduction, Turing submachines are just another name for total Turing machines. Anyway, Turing submachines can always be subsystems of either abstract universal Turing machines or of powerful (big) enough everyday computers (which are also some sort of total Turing machine, i.e. a universal Turing machine with limited resources).

Note that the class of all submachines is infinite, but not recursive.

When we talk of **subprograms** we refer to programs run on a Turing submachine. Herein, only submachines of a particular subclass will be dealt with: submachines defined by a computation time limited by a computable function². In fact, both these and the more generic submachines defined above are equivalent in computational power. To demonstrate this, just note that if a program computes a total function, then there is a program that can compute the computation time of this first program. Therefore, for every computable and total function, there is a submachine with limited computation time capable of computing this function – and, possibly, other functions as well. The reverse follows from the definition of submachine.

Let P_T be an arbitrary program that calculates a computation time for a given program w . That is, let P_T be an arbitrary total computable function. Thus, there is a **Turing submachine U_{P_T} defined by the computation-time function P_T** .

We then define submachine $U/P_{SM} \circ P_T$ (which will be a program running on U that computes a total function), where P_{SM} is a program that receives P_T and w as inputs, runs $U(P_T \circ w)$ and returns:

- (i) l_1 , if $U(w)$ does not halt within computation time $\leq U(P_T \circ w)$;
- (ii) l_{k+1} , if $U(w)$ halts within computation time $\leq U(P_T \circ w)$ e $U(w) = l_k$;

This program defines a Turing submachine that returns a known symbol (in this case, zero) when program w does not halt in time $\leq U(P_T \circ w)$ or returns the same output (except for a trivial bijection) as $U(w)$ when the latter halts in time $\leq U(P_T \circ w)$.

To be a Turing submachine, $U/P_{SM} \circ P_T$ must be defined for all inputs. This occurs because P_T is total by definition. In addition, as computation time P_T becomes more increasing, the more submachine $U/P_{SM} \circ P_T$ approaches the universality of U .

Therefore, we will denote only as U_{P_T} a Turing submachine $U/P_{SM} \circ P_T$, so that:

² Or time-bounded Turing machines

$$\forall w \in L \ (\ U_{P_T} (w) = U / P_{SM} \circ P_T (w) = U (P_{SM} \circ P_T \circ w))$$

5. FUNCTION $BB^{+}_{P'_T}(N)$

Let P'_T be a total function. Let us define function $BB^{+}_{P'_T}(N)$, which we will call Busy Beaver Plus, through the following recursive procedure:

- (i) Generate a list of all outputs of $U_{P'_T}(w)$ such that $|w| \leq N$;
- (ii) Take the largest number on that list;
- (iii) Add 1;
- (iv) Return that value.

The name of this function refers to the Busy Beaver $BB(N)$ function and, consequently, it is no coincidence that the two have almost the same definition. If step (iii) is removed, it becomes exactly the Busy Beaver function for Turing submachines, here denoted as $BB_{P'_T}(N)$. Thus:

$$BB^{+}(N) = BB(N) + 1$$

and

$$BB^{+}_{P'_T}(N) = BB_{P'_T}(N) + 1$$

But why use function BB^{+} instead of BB ? This might be, one supposes, the reader's first and immediate question. As we are dealing with Turing submachines and P'_T is arbitrary, it is possible there is a program on $U_{P'_T}$ with size $\leq N$ such that computes the highest value returned by any other program on $U_{P'_T}$ with size $\leq N$. When dealing with a universal Turing machine U , this cannot occur – except for a constant. However, with submachines, it can. Thus, function $BB^{+}_{P'_T}$ is triggered to assure it, in itself, is not relatively computable - or compressible – by any program on $U_{P'_T}$, although it can be by a program on U . Since P'_T is a program that computes a total function, then $BB^{+}_{P'_T}(N)$ is computable.

The Busy Beaver contains the idea of the greatest output of any $\leq N$ sized program; so the Busy Beaver Plus function contains the idea of increasing, at least by 1, any $\leq N$ sized program. Respectively, the first gives us maximization, and the second, an “almost” minimal increment.

Following this line of thought, to symbolize this new function, the image may be evoked of the proverbial man sitting on a hungry donkey and driving the animal by a carrot hanging from a fishing rod. As the carrot looms in front of the donkey's face, the hungry donkey is driven to walk forward to reach the carrot, which is never reached. Not because it is an infinite distance away, but because with every step it takes the carrot moves forward along with it. The carrot is always “one step” ahead of the donkey. No matter how dutifully the donkey walks toward the carrot, it will always remain at the same distance, just beyond reach, unattainable. No matter how rapidly increasing is the function P'_T , the program on U that computes $BB^{+}_{P'_T}(N)$ simply bases itself on the $U_{P'_T}$ outputs to overcome them by a minimum. No matter how powerful $U_{P'_T}$ may be, $BB^{+}_{P'_T}(N)$ will always be “one

step” ahead of the best that any subprogram (i.e., any program $U_{P'_T}$) can do.

It is worthy of note that, analogously to the Busy Beaver, the $BB^{+}_{P'_T}(N)$ may be used to measure the “creativity” or “*sub-algorithmic complexity*” of the subprograms in relation to Turing submachine $U_{P'_T}$. Why? By its very definition, if a subprogram generates an output $\geq BB^{+}_{P'_T}(N)$, it must necessarily be of size $> N$. It needs to have over N bits of relatively incompressible information, i.e. over N bits of relative creativity.

Of course, one may always build a program that will compute function $BB^{+}_{P'_T}(N)$, if function P'_T is computable. This would allow a far smaller program than N there to exist – e.g., of size $\leq C + \log_2 N + (1 + \epsilon)\log_2(\log_2 N)$ – that will compute $BB^{+}_{P'_T}(N)$. But that does not constitute a contradiction, because this program can never be a subprogram of $U_{P'_T}$, in other words, it can never be a program that runs on the computation time determined by P'_T . If it was, it would enter into direct contradiction with the definition of $BB^{+}_{P'_T}$: the program P'_T will become undefined for an input, which by assumption is false. Then, as we will show in item 6, we will immediately get the “paradox” of computability.

6. SUB-UNCOMPUTABILITY: RECURSIVE RELATIVE UNCOMPUTABILITY

Now we will prove the crucial, yet simple, result that governs this paper.

Let P'_T be a total function and $U_{P'_T}$ a Turing submachine. Then, we can prove that function $BB^{+}_{P'_T}(N)$ is **relatively uncomputable** by any program on $U_{P'_T}$. Or: there is no subprogram that, for every input N , returns an output equal to $BB^{+}_{P'_T}(N)$. Actually, $BB^{+}_{P'_T}(N)$ eventually dominates any program on $U_{P'_T}$.

A more intuitive way to understand what is going on is to look for a program and concatenate its input, such as $U_{P'_T}(P * N)$ for instance. Where “*” denotes the optimal functionalizing concatenation, and not necessarily the “concatenation” “o” defined in item 2. In fact, this applies to any way to compress the information of P and N in an arbitrary subprogram. Therefore, it may not be in the form $P \circ N$.

We avail ourselves of the same idea used in the demonstration of Chaitin’s incompleteness theorem. Now, however, to demonstrate an uncomputability relative to the submachine $U_{P'_T}$.

When N is given as input to any program P , it comes in its compressed form with size $\cong H(N)$ – in fact, we use the property

$$|P \circ N| \leq C + |P| + C' + \log_2 N + (1 + \epsilon)\log_2(\log_2 N)$$

whereby $|P \circ N| \cong C + H(N)$. But, as already known by the AIT, for any constant C there is a big enough N_0 such that $C + H(N_0) < N_0$. Therefore, according to the definition of $BB^{+}_{P'_T}$, the output of $P \circ N_0$ when run on submachine $U_{P'_T}$, will be taken into account when one calculates $BB^{+}_{P'_T}(N_0)$. Thus, necessarily,

$$BB^{+}_{P'_T}(N_0) \geq U_{P'_T}(P \circ N_0) + 1 > U_{P'_T}(P \circ N_0)$$

Which will lead to contradiction, if P computes $BB^{+}_{P'_T}$ when running on submachine $U_{P'_T}$. The same holds for “*”.

Also, following the same argument, it can be shown promptly that $BB^{+_{P'_T}}(N)$ is a **relatively incompressible, or sub-incompressible**, function by any subprogram smaller than or equal to N . That is, no program of size $\leq N$ running on P'_T will result in an output larger than or equal to $BB^{+_{P'_T}}(N)$.

7. CONCLUSION AND FINAL COMMENTS

First, a self-delimiting universal language L was defined for a universal Turing machine U . Then, we defined the Turing submachines (or total Turing machines) $U_{P'_T}$. It has been further demonstrated that the phenomenon of “sub-uncomputability” is ubiquitous: for every Turing submachine $U_{P'_T}$, if is a program that computes a total function, then the computable function $BB^{+_{P'_T}}(N)$ is relatively uncomputable by any program running on $U_{P'_T}$ in the same manner that the Busy Beaver function $BB'(N)$ is in relation to any program. Also, by the very definition of $BB^{+_{P'_T}}$, there cannot be any program of size $\leq N$ running on $U_{P'_T}$ that will generate an output higher than or equal to $BB^{+_{P'_T}}(N)$ – which may be called the sub-incompressibility of the function $BB^{+_{P'_T}}$.

To recreate/relativize the classic uncomputability of the Busy Beaver function, essentially, we had to do to Turing and Radó the same as Skolem did to Cantor: we relativized the uncomputability of function $BB'(N)$. It was demonstrated that it depends on “being seen from the outside, or from the inside”. Thus, it was proven that there is a “paradox” of computability *a la* Skolem, i.e. there is a function that is computable if “seen from without”, that is uncomputable if “seen from within”. As both language L and the submachines can be programmed, this phenomenon can occur within our everyday computers.

However, not “all uncomputabilities” of a first-order hypercomputer were relativized in relation to a universal Turing machine. Only what was described above was relativized. However, following this line of mathematical inquiry enabled us to build metabiological evolutionary models that are fully analogous to Chaitins models of Intelligent Design and Cumulative Evolution – as shown in [4]. For this purpose, a Turing submachine $U_{P^{**_T} \circ P_T}$ was built and a relative and computable Chaitin Omega number $\Omega_{P^{**_T} \circ P_T}$ in the case, a time-limited halting probability was defined. Clauses were added to $U_{P^{**_T} \circ P_T}$ to allow the existence of finite lower approximations ρ to $\Omega_{P^{**_T} \circ P_T}$ that can be used by a program P when running on $U_{P^{**_T} \circ P_T}$ to compute values of $BB^{+_{P^{**_T} \circ P_T}}(N)$, so that $2N + C \geq |P \circ \rho| \geq N + 1$, where C is a constant. This was also designed to mimic what a universal Turing machine can do with lower approximations to Ω with the purpose of calculating values of $BB'(N)$. Also, another clauses were added to enable the relative versions of key mutations/programs from Chaitins models to also become subprograms. Thus, the open-ended evolution of subprograms revealed itself as isomorphically fast as the open-ended evolution of programs. It allowed us to recursively relativize more “behaviors” of a first order hypercomputer in relation to a computer, making them happen between machines and submachines.

An upcoming mathematical inquiry this article suggests is proving whether or not there is a way to define - relatively - universal Turing submachines. A

universal submachine should be analogous to a universal machine, so there is a class of subcomputable problems, always reducible by subprograms (in the case to the above, within a subcomputable time), that are computable by this universal submachine. The questions would be: how to define a computation time P_U so that U_{P_U} is a universal Turing submachine? Is it possible? This mathematical problem also involves studying the greatest amount of first order uncomputable functions that can be relativized to become sub-uncomputable. If this Turing submachine is possible, a negative Turing degree can be defined. Moreover, as for each Turing submachine $U_{P'_T}$ there is always another more powerful and non-reducible submachine $U_{P''_T}$ such that P''_T is sufficient computation time to compute $BB^+_{P'_T}$, so it would likewise be possible to create an infinite hierarchy of negative Turing degrees.

To what extent can a computer be made to “behave”, in relation to one of its subcomputers, as if it was a hypercomputer?

REFERENCES

- [1] Abrahão, F. S. (2011). Questões em Metabiologia, *Scientiarum Historia*.
- [2] Abrahão, F. S. (2013). Metabiologia Cantoriana, *Scientiarum Historia*.
- [3] Abrahão, F. S. (2014). “Paradoxo” da Computabilidade, *Scientiarum Historia*.
- [4] Abrahão, F. S. (2015). *Metabiologia, Subcomputação e Hipercomputação: em direção a uma teoria geral de evolução de sistemas*, Ph.D. thesis, Federal University of Rio de Janeiro.
- [5] Barwise, J. (ed.) (1977). *Handbook of Mathematical Logic* (Elsevier Science Publisher B.V.).
- [6] Basu, S. (1970). On the structure of subrecursive degrees, *Journal of Computer and System Sciences*.
- [7] Burgin, M. (2005). Measuring power of algorithms, programs, and automata, *Artificial Intelligence and Computer Science*, pp. 1–61.
- [8] Calude, C. (ed.) (2007). *Randomness and Complexity* (World Scientific).
- [9] Chaitin, G. (2012). *A Computable Universe: Understanding and Exploring Nature as Computation*, chap. Life as Evolving Software (World Scientific), pp. 277–302.
- [10] Chaitin, G. (2013). *Proving Darwin: making biology mathematical* (Vintage Books).
- [11] Chaitin, G., Chaitin, V., and Abrahão, F. S. (2014). Metabiología: los orígenes de la creatividad biológica, *Investigación y Ciencia*, p. 448.
- [12] Enderton, H. (2001). *A Mathematical Introduction to Logic* (Academic Press).
- [13] Ewert, W., Dembski, W., and Marks II, R. J. (2013). Active information in metabiology, *Bio-Complexity*.
- [14] Lewis, H. R. and Papadimitriou, C. H. (2000). *Elementos de Teoria da Computação* (Bookman).
- [15] Rogers, H. (1992). *Theory of Recursive Functions and Effective Computability* (MIT Press).
- [16] Rose, H. (1987). Subrecursion: Functions and hierarchies, *The Journal of Symbolic Logic*.
- [17] Zenil, H. (ed.) (2011). *Randomness through Computation* (World Scientific).
- [18] Zenil, H. (ed.) (2013). *A Computable Universe* (World Scientific).